ソース説明

・HeatUp

このプロジェクトは Unity で制作されています。

1_Character

すべての敵キャラのベースとなるクラス。

体力の初期値や基本的な敵キャラクターの機能をすべて保有している。

```
public class Character : MonoBehaviour
{
    /// <summary>
    /// 体力の初期値
    /// </summary>
    [Header("Character")]
    [TooltipAttribute("体力の初期値")]
    public int maxHp = 100;

    /// <summary>
    /// 地面に着地しているかどうかを判断するためのオフセット
    /// </summary>
    [TooltipAttribute("地面に着地しているかどうかを判断するためのオフセット")]
    public Vector2 groundCheckOffset = new Vector2(0, -1);

    /// <summary>
    /// HealthBarの位置
    /// </summary>
    [TooltipAttribute("HealthBarの位置")]
    public Vector2 healthBarOffset = new Vector2(0, 2);

    /// <summary>
    /// 同じキャラクター同志の当たり判定とプレイヤーとの当たり判定を無視する
    /// </summary>
    [TooltipAttribute("同じキャラクター同志の当たり判定とプレイヤーとの当たり判定を無視する")]
    public bool ignoreCollision = false;

    /// <summary>
    /// IgnoreColliderの大きさ
    /// </summary>
    [TooltipAttribute("IgnoreColliderの大きさ")]
    public float ignoreSize = 5;

    /// <summary>
    /// プレイヤーのhp
    /// </summary>
```

上記の画像のようにまとめておくことにより、大量のキャラクターを同じベースプログラムに沿わせ、新たな機能の追加ができるように設計。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SubCharacter : Character
{
    protected override void Start()
    {
        base.Start();//消さないでください。

        //ここから処理を書いてください。
    }

    protected override void Update()
    {
        base.Update();//消さないでください。

        //ここから処理を書いてください。
    }


    //キャラクターのhpが0になった時にDestroy(gameObject)以外の処理を書きたい場合はコメントを解除してください。
    /*
    protected override void DestroyCharacter()
    {
        Debug.Log("A");
        //base.DestroyCharacter();//Destroyしたくない場合は消去する。
    }
    */
}
```

上記の画像のように Character クラスの継承としてクラスを作成することにより他のプログラマーとの複数キャラクター制作の効率化を試みた。

2_Player
プレイヤーの機能をすべて管理するクラス

3_Sword
剣の処理を行うクラス

4_Bridge
橋の処理

5_CameraController
カメラの移動を統括するクラス

6_CameraShake
カメラに振動を与えるクラス

7_CameraTrigger
カメラが強制的に移動するイベントを発生させるクラス

8_ColliderAttacher
Character にコライダーをアタッチするクラス

9_Fade
簡易的にフェードイン、フェードアウトをさせる処理

10_FollowCamera
カメラをプレイヤーを追尾させるクラス

11_GameClear
ゲームをクリアしたときのイベントを発生させるクラス

12_GameClearCollider
ゲームのクリア判定を当たり判定を用いて判断するクラス

13_Gizmos2
Unity 内で制作をするにあたり必要である補正線などの追加

14_HealthBar
敵のヘルスバーの処理をするクラス

15_IgnoreCollision
すべての敵同士がコリジョンをを一気に無視すると重くなるので、近くにいる敵の当たり判定だけを無視する処理

16_Ladder
プレイヤーが登れる梯子の処理をするクラス

17_MoveDirection
奥の階層に行くときに、ある一定の距離まで勝手に進むようにするための向きを保有するクラス

18.Plant
敵を倒すと出てくる水蒸気を利用して、成長するつるの処理をするクラス

19_ReSpawn
ヒットした場合 RespawnTrigger にプレイヤーを移動させるクラス

20_ReSpawnByDestroy
ゲームオブジェクトが消滅したらリスポーンさせるクラス

21_ReSpawnTrigger
リスポーンする位置を更新する Trigger（ある一定の場所まで行けば、リスポーンする位置を変えるクラス）

22_ShowStart
リスタート時にスタート画面を表示するかどうかを保存するクラス

23_SpawnArea
敵をスポーンさせるエリアの処理をするクラス

24_Spawner
プレイヤーがある一定のエリアに入った時に敵を出現させるトリガーとなるクラス

25_SpineAnimation

スパインのアニメーションをアニメーターで使うための処理

26_SpriteRender3D
Sprite を 3D のポリゴンとして描画するクラス

27_SubCharacter
Character クラスをもとに制作したプログラマ制作用のテンプレート

・ Refleconnect

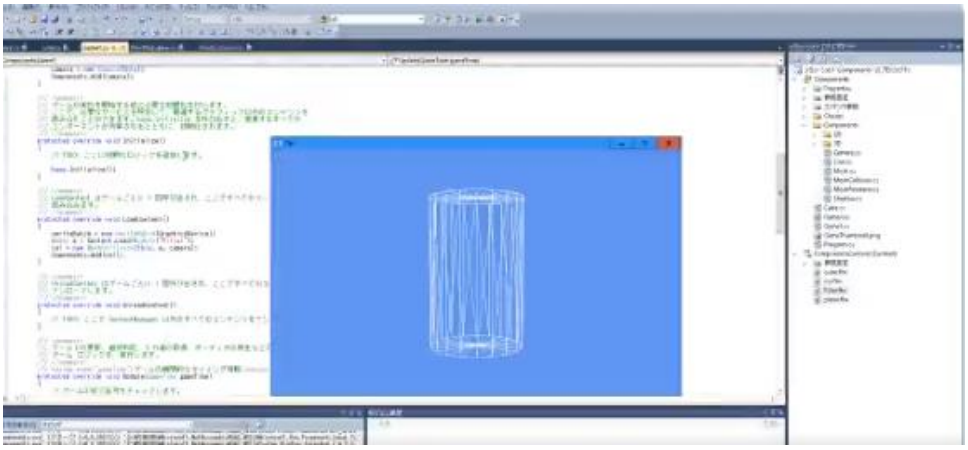
このプロジェクトは XNA で制作されています。大学で初めゲームを作った際のものであり、すべてのプロジェクトの中で一番古いものになります。 一つのポリゴンを表示させるにも 100 行ほど書かなくてはならないなど、他のプログラマーとのソースの共有が難しい制作であったため、ライブラリを制作しました。 Refleconnect 内のソースは当時制作したライブラリの中身も含まれています。プログラムリーダーとなったため、ライブラリを制作しできる限りまとめたつもりですが、今見ると粗が目立っているのがよくわかります。

1_Transform
オブジェクトの位置、回転、スケールなどの数値情報を保有するクラス

2_MeshRenderer
メッシュを描画するクラス

3_Mesh
メッシュに必要なマテリアルなどを含めたクラス

4_MeshCollision
メッシュから当たり判定を作るクラス。頂点情報をもとに、三角ポリゴンの当たり判定を行うという。当時作り上げたライブラリの中で最も難易度が高かったもの。



5_ParticleSystem
パーティクルを生成するクラス

6_Button
ボタンを制御するクラス

7_ButtonMaster
ボタンを統括し、一番近いボタンへの遷移を可能にするクラス

8_Camera
カメラの処理をするクラス

9_CharacterMaster
メインキャラクターを統括するクラス

10_MainCharacter
メインキャラクターの処理を行うクラス

11_Particle
ParticleSystem で生成されるパーティクル

12_Sphere
プレイヤー間でパスするボールのクラス

13_Boss
ボスのクラス

14_Calculation
ゲームで使用する計算関数をまとめたクラス

15_Slime
スライムのクラス

16_Slimes
スライムを統括するクラス

17_Stage
ステージの基礎となるクラス

18_Stage1
Stage をもとに作られた Stage1

19_Stage2
Stage をもとに作られた Stage2

20_Stage3
Stage をもとに作られた Stage3

・ Triad

このプロジェクトは Unity で制作されています。AR と声を合わせたゲームです。ネットワーク通信を使い音声データを共有するなどの機能があり、あまり見たことのないゲームに仕上がったため、大学ならではの制作ができたと思いました。

1_Goddess
女神像のクラス

2_GoddessNetwork
女神像のネットワーク処理を行うクラス

```
public void SendAudio(int index, AudioClip clip)
{
    if (isServer)
    {
        goddess.AddClip(index, clip);
    }

    //音声データ取得
    float[] d = new float[clip.samples * clip.channels];
    clip.GetData(d, 0);

    //音声データが非常に大きいため、分割して送信する
    int sendLoop = d.Length / PART_SIZE + (d.Length % PART_SIZE > 0 ? 1 : 0);
    for (int i = 0; i < sendLoop; i++)
    {
        int start = i * PART_SIZE;
        int end = (i + 1) * PART_SIZE;
        if (end >= d.Length)
        {
            end = d.Length;
        }

        //分割出来たデータを送信する
        float[] part = new float[PART_SIZE];
        Array.Copy(d, start, part, 0, end - start);
        AudioData s = new AudioData()
        {
            index = index,
            part = part,
            start = start,
            end = end,
            size = d.Length,
            name = clip.name,
            channnels = clip.channels,
            frequency = clip.frequency
        };
```

音声のデータが非常に大きいため、4096 個のパーツに分けて送信する方法を採用。

3_Address
サーバー開設やルーム作成を行うためのアドレス保持クラス

4_BarrierDestruction
バリアを破壊するクラス

5_CheckNote
音程をチェックするクラス

6_Destroy
アタッチされたオブジェクトを一定時間経過すると自動破棄するクラス

7_DetectedPanelFinder
AR の平面検知に使用される DetectedPanel を探すクラス

8_GoddessUIController

女神像の UI コントローラークラス

9_LockCamera
常にカメラの方向に向かせるクラス

10_LogDisplay
ログを画面に出力させるクラス

11_NoteNameDetector
ピッチを測定するクラス

12_ShowLog
デバック用に表示できるログビューを統括するクラス

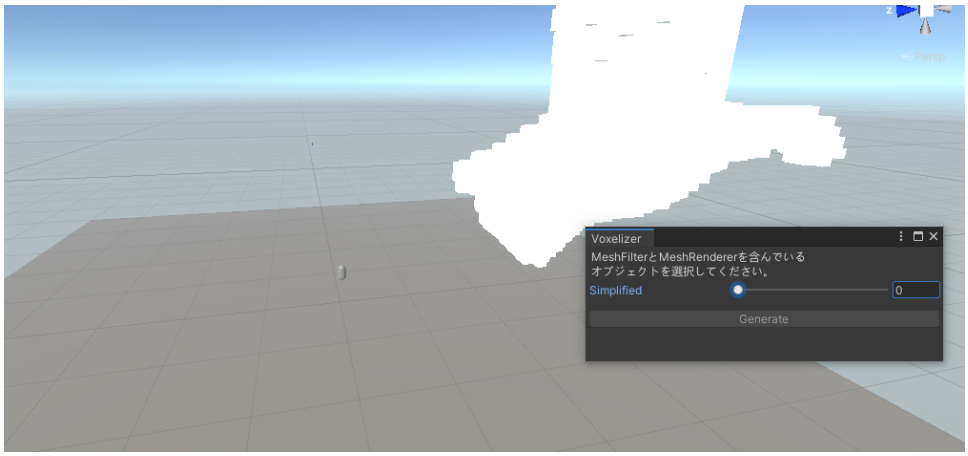
13_WeakPoint
弱点ポイントのクラス

・ Voxel

このプロジェクトは Unity で制作されています。現在制作進行中の企画の研究内容である、大量のボクセル破壊表現を可能にするベースプログラムになります。参考にしたものはマインクラフトのメッシュ描画方法です。見えていないところのメッシュは描画せず、見えている範囲のボクセルをいっぺんに描画するという方法です。

1_Chunk
ボクセルのかたまりを描画するクラス。ボクセルを一つづつ描画すると非常に重くなるため、一つのかたまりとして、オブジェクトを描画します。もちろん頂点や、UV などの割り当て処理もしなければならないので、少し手間はかかりますが、破壊表現時に 4 倍程度のフレームレートを出すことができました。Json データとして Chunk の保存もできるので、大量のボクセルデータのロードの時間の短縮も図れました。

2_Voxelizer
メッシュをボクセル化するエディタークラス



3_Editor_Chunk
Chunk のインスペクターにあるボタン用エディタークラス

4_Editor_Voxelizer
Voxelizer にあるボタン用エディタークラス

5_Voxel
破壊後に一つのボクセルとして物理挙動するボクセルクラス

6_VoxelData
ボクセル生成に使用するボクセルデータを統括したクラス

7_VoxelDestroyer
範囲外に飛ばされたボクセルを破棄するクラス

8_VoxelGenerator
ボクセルを生成するクラス